An Introduction to Data

Martin Frigaard

2021-11-14

What is/are data?

For the sake of simplicity, we're going to use the term data to describe "*something that has been measured.*" As you will see throughout the course, measurement can take many forms and encompasses just about any phenomena we can imagine. I will use the terms measurement and metrics interchangeably, which are defined as, "*a method of measuring something, or the results obtained from this.*"

Data are measurements, and measurements are approximations

Measurements and metrics are usually intended to capture things that matter (otherwise, why measure them?), but it's also important to remember that **all measurements are approximations**. It's unlikely that the question we're trying to answer has a dataset that *perfectly* captures the phenomena of interest. This course focuses on using data visualizations (graphs and tables) to display and communicate some phenomena.

In statistics, a widespread saying is, "all models are wrong, but some are useful." Models use these approximate measurements to explain or predict, and data visualizations visually display these approximations.

An example question

For example, consider the question, "*who are the best hitters in professional baseball?*" To answer this question, we need to make a few decisions about the measurements we'll be using.

First, how are we going to define "best"? Do we want to base our definition on the number of homeruns, the on-base percentage (the ratio measuring the times a player ended up on a base–whether it was by a hit, walk, or getting beaned by pitch–versus the total number of plate appearances), or some other measurement of hitting performance?

Second, we need to decide what constitutes a "professional baseball" player–should we base this on a minimum number of games played in Major League Baseball (the current organization), or include data from previous leagues?

Below are some examples of follow-up questions we might also consider:

- *Which current MLB players have the most homeruns?*
- *Who had the most RBIs (runs batted in) in the history of baseball (outside of just the MLB)?*
- *What player had the best OBP (on-base percentage) last season?*

As we can see, even a seemingly simple question requires qualifications and operational definitions. Before we can design a data visualization, we need to frame the question in a way that can be answered with the available data (or we need to collect data to answer we're interested in). Only then we can start determining what the best visualization is for displaying and communicating the results.

In our baseball example, we would need to decide on the metrics we'll use to measure 'best hitting', and the criteria for being considered a 'professional baseball player'. Then we need to rewrite our original question, but this time with more precision. This process–of re-framing a question into something we can answer with data–is essential *before we start writing code.*

We'll go over a few terms and concepts you'll commonly encounter when working with data in the next few sections.

Raw data

The 'raw data' is the data in it's 'initial' or 'original' format it was collected, received, or downloaded in. We always want to keep a copy of the raw data 'untouched' by any of the processing steps or transformations we need to perform to create data visualizations. Most raw data isn't in the format we need to create a visualization, so it's essential to document all the steps we needed to make each output.

Sources

The 'source' is the origin(s) of the raw data used in a visualization, usually a link or URL. For example, a source for our baseball question could be the Baseball Databank, an "Open Data collection of historical baseball data".

A source usually includes a data dictionary. This document describes the names and content of the variables included in each dataset.

Organizing Data

In this course, we'll be using the term 'dataset' to describe rectangular data objects (like spreadsheets). Datasets are made up of variables (columns) and observations (rows). At the intersection of each variable and observation is a value, the smallest unit of data in a dataset.

An example dataset is the People table from the Baseball Databank listed above.

The path for these data is listed below:

```
"https://raw.githubusercontent.com/chadwickbureau/baseballdatabank/master/core/People.csv"
```

We're going to dissect this url a bit, because it will help us understand how to organize data on our local machine when we're downloading it (or if someone sends it to us), and it will give us some practice programming.

URL structure

The first bit of the url contains the website, or `root`. If we look the text, it's telling us this is a 'raw' file from Github. In fact, if we click on this link, it takes us to Github.com.

Let's store the website root in a vector called `root`.

```r
root <- "https://raw.githubusercontent.com/"
root
```

```
## [1] "https://raw.githubusercontent.com/"
```

The next vector we'll create is `user_repo`, which is the 'user' and 'repo', separated by the forward slash (`/`).

You could create a repo with your Github account (which you set up to use Rstudio.Cloud) and it would look the same way (`username/repo-name`)

```r
user_repo <- "chadwickbureau/baseballdatabank/"
user_repo
```

```
## [1] "chadwickbureau/baseballdatabank/"
```

Github is a version control system, and it functions by using branches. The default branch is usually referred to as the 'master' or 'main' branch.

Since we're downloading data from the `core` folder of the `master` branch, we will create a vector called `folder_branch` that contains `master/core/`

```r
folder_branch <- "master/core/"
folder_branch
```

```
## [1] "master/core/"
```

Finally, at the end of the url is the file name, 'People.csv'. We will store this in the `dataset_name` vector.

```r
dataset_name <- "People.csv"
dataset_name
```

```
## [1] "People.csv"
```

Creating data folders in R

Since we're downloading a raw data file, we should create a folder on our local machine called `data`, and a subfolder called `raw`.

```r
raw_data_folder <- "data/raw/"
raw_data_folder
```

```
## [1] "data/raw/"
```

We can create this folder using `fs::dir_create()`.

```r
fs::dir_create(raw_data_folder)
```

The `dir_create()` function is great because it will check to see if the folder exists, and won't overwrite it if it does!

Below is a folder tree with the `data/raw` folders listed:

```
|-- data/
    |-- raw/
```

Downloading data with R

R comes with a function for downloading files, `download.file()`, which we will use to download the `People.csv` file. The `download.file()` function takes two arguments, `url =` and `destfile =`.

Recall that the original url was the combination of `root`, `user_repo`, `folder_branch` and `dataset_name`.

We can practice a little programming here by using the `paste0()` function to recreate the link to the dataset on Github, which we will store in `people_url`.

```r
people_url <- paste0(root, user_repo, folder_branch, dataset_name)
people_url
```

```
## [1] "https://raw.githubusercontent.com/chadwickbureau/baseballdatabank/master/core/People.csv"
```

The `destfile` (destination file) will be the local destination of the file, `data/raw/People.csv`.

```r
people_file <- "data/raw/People.csv"
people_file
```

```
## [1] "data/raw/People.csv"
```

Now we can use `people_url` and `people_file` to download the 'People.csv' from Github into our local `data/raw` folder.

```
download.file(url = people_url, destfile = people_file)
```

The output from `download.file()` should look something like this:

```
trying URL 'https://raw.githubusercontent.com/chadwickbureau/baseballdatabank/master/core/People.csv'
Content type 'text/plain; charset=utf-8' length 2669722 bytes (2.5 MB)
==================================================
downloaded 2.5 MB
```

Below is an updated folder tree with the downloaded 'People.csv' dataset.

```
|-- data/
    |-- raw/
        |-- People.csv
```

Importing data into R

We now have the 'People.csv' data downloaded into a local folder, we can import the file into RStudio using one of R's many import functions.

For this example, we'll use the `read_csv()` function from the readr package. `readr` was designed to '*provide a fast and friendly way to read rectangular data*', so it's perfect for importing 'People.csv'.

`read_csv()` takes a `file` argument, with we can specify with our `people_file` vector. We will store these data in a data object named `RawPeople`.

```
install.packages("readr")
library(readr)
readr::read_csv(file = people_file)
```

Rectangular data (`tibbles`)

I've re-written the `read_csv()` function below with the explicit data path. Before we import these data, we should know what to expect. A quick read of the documentation (accessible using `??readr::read_csv` in the console) tells us the value (output) from `read_csv()` will be a `tibble`. Recall what we learned about `tibbles`/`data.frames` in the Intro to R Programming: Rectangular data in R exercises.

We're going to import the 'People.csv' file into an object named `RawPeopleTbl`.

```
RawPeopleTbl <- read_csv(file = "data/raw/People.csv")
```

```
## Rows: 20358 Columns: 24

## -- Column specification ------------------------------------------------------
## Delimiter: ","
## chr  (14): playerID, birthCountry, birthState, birthCity, deathCountry, deat...
## dbl   (8): birthYear, birthMonth, birthDay, deathYear, deathMonth, deathDay,...
## date  (2): debut, finalGame
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

When we run this function, we can see it gave us some useful information about the file we imported. We can see the number of columns (24) and rows (20322). In the section titled, "`Column specification`", we can see the format (`chr`, `dbl`, and `date`) and name (`playerID`, `birthYear`, and `debut`) of the columns (for a refresher on these data types, refer to data types slides)

Rectangular data (`data.frames`)

Just for comparison, we'll compare `readr`'s `read_csv()` to the `read.csv()` function from the `utils` package (loaded 'out of the box' with R). The documentation (accessible using `??utils::read.csv` in the console) tells us the
output object will be a `data.frame`, so we will store this in an object named `RawPeopleDf`.

```r
RawPeopleDf <- read.csv(file = "data/raw/People.csv")
```

The `utils::read.csv()` function does not give us the same output as `readr::read_csv()`, but we can check the structure of this object using `utils::str()`.

```r
utils::str(RawPeopleDf)
```

```
## 'data.frame':    20358 obs. of  24 variables:
##  $ playerID    : chr  "aardsda01" "aaronha01" "aaronto01" "aasedo01" ...
##  $ birthYear   : int  1981 1934 1939 1954 1972 1985 1850 1877 1869 1866 ...
##  $ birthMonth  : int  12 2 8 9 8 12 11 4 11 10 ...
##  $ birthDay    : int  27 5 5 8 25 17 4 15 11 14 ...
##  $ birthCountry: chr  "USA" "USA" "USA" "USA" ...
##  $ birthState  : chr  "CO" "AL" "AL" "CA" ...
##  $ birthCity   : chr  "Denver" "Mobile" "Mobile" "Orange" ...
##  $ deathYear   : int  NA 2021 1984 NA NA NA 1905 1957 1962 1926 ...
##  $ deathMonth  : int  NA 1 8 NA NA NA 5 1 6 4 ...
##  $ deathDay    : int  NA 22 16 NA NA NA 17 6 11 27 ...
##  $ deathCountry: chr  "" "USA" "USA" "" ...
##  $ deathState  : chr  "" "GA" "GA" "" ...
##  $ deathCity   : chr  "" "Atlanta" "Atlanta" "" ...
##  $ nameFirst   : chr  "David" "Hank" "Tommie" "Don" ...
##  $ nameLast    : chr  "Aardsma" "Aaron" "Aaron" "Aase" ...
##  $ nameGiven   : chr  "David Allan" "Henry Louis" "Tommie Lee" "Donald William" ...
##  $ weight      : int  215 180 190 190 184 235 192 170 175 169 ...
##  $ height      : int  75 72 75 75 73 74 72 71 71 68 ...
##  $ bats        : chr  "R" "R" "R" "R" ...
##  $ throws      : chr  "R" "R" "R" "R" ...
##  $ debut       : chr  "2004-04-06" "1954-04-13" "1962-04-10" "1977-07-26" ...
##  $ finalGame   : chr  "2015-08-23" "1976-10-03" "1971-09-26" "1990-10-03" ...
##  $ retroID     : chr  "aardd001" "aaroh101" "aarot101" "aased001" ...
##  $ bbrefID     : chr  "aardsda01" "aaronha01" "aaronto01" "aasedo01" ...
```

These objects are similar, but have a few important differences. First, `tibble`s have their own `class()`:

```
class(RawPeopleTbl)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
class(RawPeopleDf)
```

```
## [1] "data.frame"
```

This tells us that a `tibble` is a `data.frame`, but a `data.frame` is not a `tibble`.

Differences (and similarities) between `tibble`s and `data.frame`s
Knowing if your data is stored as a tibble or data.frame is important because if we're going to extract (subset) columns or rows from a rectangular data object in R, what we get back depends on the type of object we have.

For example, if we want to subset the `nameLast` for the second row in 'People.csv', and it's stored in a `tibble` (like `RawPeopleTbl`), we can do this using `tibble[ row number, column number]` syntax.

If you need a refresher on subsetting, see the methods discussed in Subsetting R Objects from week 2

```
RawPeopleTbl[ 2, 15]
```

```
## # A tibble: 1 x 1
##   nameLast
##   <chr>
## 1 Aaron
```

We can see this returns a tibble with the last name for Hank Aaron. If we want to do the same thing to a `data.frame` (like we did above), we can see the output is a vector (not a `tibble` or `data.frame`).

```
RawPeopleDf[ 2, 15]
```

```
## [1] "Aaron"
```

The difference in behavior is because the return object from the `data.frame` above is a single element (not two dimensions), so it defaults to a vector (`tibble`s always return `tibble`s).

However, if the return object has two dimensions (i.e. a row and a column), then the subsetting behavior is nearly identical. See below when we try to subset the `nameFirst` and `nameLast` for Hank Aaron (row 2).

```r
# tibble return
RawPeopleTbl[ 2, 14:15]
```

```
## # A tibble: 1 x 2
##   nameFirst nameLast
##   <chr>     <chr>
## 1 Hank      Aaron
```

```r
# data.frame return
RawPeopleDf[ 2, 14:15]
```

```
##   nameFirst nameLast
## 2      Hank    Aaron
```

We're going to be primarily using `tibbles` throughout the course because they print more information to the screen (and are a bit more predictable).

Variables

Variables are the columns in a dataset. Sometimes you'll see them referred to as attributes, predictors, outcomes, responses, or features.

We can check the column names from `RawPeople` using `colnames()` or `names()`.

```r
colnames(RawPeopleTbl)
```

```
##  [1] "playerID"     "birthYear"    "birthMonth"   "birthDay"     "birthCountry"
##  [6] "birthState"   "birthCity"    "deathYear"    "deathMonth"   "deathDay"
## [11] "deathCountry" "deathState"   "deathCity"    "nameFirst"    "nameLast"
## [16] "nameGiven"    "weight"       "height"       "bats"         "throws"
## [21] "debut"        "finalGame"    "retroID"      "bbrefID"
```

```r
names(RawPeopleDf)
```

```
##  [1] "playerID"     "birthYear"    "birthMonth"   "birthDay"     "birthCountry"
##  [6] "birthState"   "birthCity"    "deathYear"    "deathMonth"   "deathDay"
## [11] "deathCountry" "deathState"   "deathCity"    "nameFirst"    "nameLast"
## [16] "nameGiven"    "weight"       "height"       "bats"         "throws"
## [21] "debut"        "finalGame"    "retroID"      "bbrefID"
```

Standardizing variable names

One of the first things we can do to make `RawPeopleTbl` easier to program with is standardize the column names. As we can see from the output above, the column names follow a camel case format. We're going to convert the names to snake case because it converts all the letters to lowercase (which reduces the chances of typos). The janitor package has a very convenient function for performing this operation, `janitor::clean_names()`.

When we apply this function, we want to store the output in an object called `PeopleTbl` (because it's no longer `RawPeopleTbl`).

```
PeopleTbl <- janitor::clean_names(RawPeopleTbl, case = "snake")
```

We will view the contents of `PeopleTbl` with the `glimpse()` function from `pillar`.

```
glimpse(PeopleTbl)
```

```
## Rows: 20,358
## Columns: 24
## $ player_id     <chr> "aardsda01", "aaronha01", "aaronto01", "aasedo01", "abad~
## $ birth_year    <dbl> 1981, 1934, 1939, 1954, 1972, 1985, 1850, 1877, 1869, 18~
## $ birth_month   <dbl> 12, 2, 8, 9, 8, 12, 11, 4, 11, 10, 9, 3, 10, 2, 8, 9, 6,~
## $ birth_day     <dbl> 27, 5, 5, 8, 25, 17, 4, 15, 11, 14, 20, 16, 22, 16, 17, ~
## $ birth_country <chr> "USA", "USA", "USA", "USA", "USA", "D.R.", "USA", "USA",~
## $ birth_state   <chr> "CO", "AL", "AL", "CA", "FL", "La Romana", "PA", "PA", "~
## $ birth_city    <chr> "Denver", "Mobile", "Mobile", "Orange", "Palm Beach", "L~
## $ death_year    <dbl> NA, 2021, 1984, NA, NA, NA, 1905, 1957, 1962, 1926, NA, ~
## $ death_month   <dbl> NA, 1, 8, NA, NA, NA, 5, 1, 6, 4, NA, 2, 6, NA, NA, NA, ~
## $ death_day     <dbl> NA, 22, 16, NA, NA, NA, 17, 6, 11, 27, NA, 13, 11, NA, N~
## $ death_country <chr> NA, "USA", "USA", NA, NA, NA, "USA", "USA", "USA", "USA"~
## $ death_state   <chr> NA, "GA", "GA", NA, NA, NA, "NJ", "FL", "VT", "CA", NA, ~
## $ death_city    <chr> NA, "Atlanta", "Atlanta", NA, NA, NA, "Pemberton", "Fort~
## $ name_first    <chr> "David", "Hank", "Tommie", "Don", "Andy", "Fernando", "J~
## $ name_last     <chr> "Aardsma", "Aaron", "Aaron", "Aase", "Abad", "Abad", "Ab~
## $ name_given    <chr> "David Allan", "Henry Louis", "Tommie Lee", "Donald Will~
## $ weight        <dbl> 215, 180, 190, 190, 184, 235, 192, 170, 175, 169, 210, 1~
## $ height        <dbl> 75, 72, 75, 75, 73, 74, 72, 71, 71, 68, 73, 71, 70, 78, ~
## $ bats          <chr> "R", "R", "R", "R", "L", "L", "R", "R", "R", "L", "R", "~
## $ throws        <chr> "R", "R", "R", "R", "L", "L", "R", "R", "R", "L", "R", "~
## $ debut         <date> 2004-04-06, 1954-04-13, 1962-04-10, 1977-07-26, 2001-09~
## $ final_game    <date> 2015-08-23, 1976-10-03, 1971-09-26, 1990-10-03, 2006-04~
## $ retro_id      <chr> "aardd001", "aaroh101", "aarot101", "aased001", "abada00~
## $ bbref_id      <chr> "aardsda01", "aaronha01", "aaronto01", "aasedo01", "abad~
```

An important this to remember about `tibbles` and `data.frames` in R is that they are both lists, but their columns are vectors of the same type and equal length.

Observations

Observations are the rows in a dataset. Sometimes you'll see these referred to as cases, records, instances, or samples. The number of rows is printed with `glimpse()` (`Rows: 20,322`), but we can check the number of rows in a dataset using the `nrow()`.

```
nrow(PeopleTbl)
```

```
## [1] 20358
```

Another method of subsetting data by rows is using the `base::subset()` function. This function takes an `x` (the object to subset), logical conditions (something that evaluates to `TRUE` or `FALSE`), and the columns to return.

Below is an example using subset to extract Hank Arron's first and last name from `PeopleTbl`.

```
subset(x = PeopleTbl,
       name_last == "Aaron" & name_first == "Hank",
       name_first:name_last)
```

```
## # A tibble: 1 x 2
##   name_first name_last
##   <chr>      <chr>
## 1 Hank       Aaron
```

We will cover other methods for row-wise operations more in the data manipulation exercises and slides.

Exporting data fromm Rstudio

Assuming we won't be making any additional changes to the `PeopleTbl` dataset, we're going to export it into the `data/` folder. We can do this with the `readr::write_csv()` function.

`write_csv()` takes two arguments, `x` (the name of the object to export), and `file` (the name and location of the output file, including the file extension).

```
write_csv(x = PeopleTbl, file = "data/PeopleTbl.csv")
```

Below is a folder tree of the `data` folder with the original raw data, and the `PeopleTbl` dataset with the changed the column names.

```
data/
  |-- PeopleTbl.csv
  |-- raw
      |-- People.csv
```

In closing

We've covered the general concept of data and measurements, how to import data onto your local machine, and keeping raw data separate from any processed datasets.

We imported a downloaded data file into RStudio, made some changes to this dataset, then exported it back onto our computer for safe keeping.

Keep this file for future reference when we're discussing data import/export.